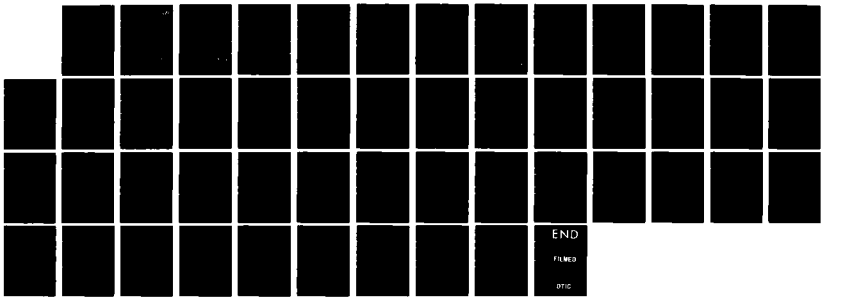
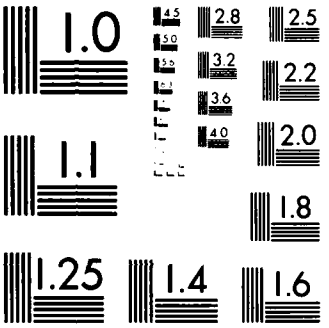
AD-A149 271 ADA IN MISSION CRITICAL SYSTEM ACQUISITION: A GUIDEBOOK 1/1
(U) MITRE CORP MCLEAN VA MITRE C3I DIV S N MOHANTY
SEP 84 MTR-84W00189 F19628-84-C-0001

UNCLASSIFIED F/G 9/2 NL





Ada in Mission Critical System Acquisition: A Guidebook

MITRE

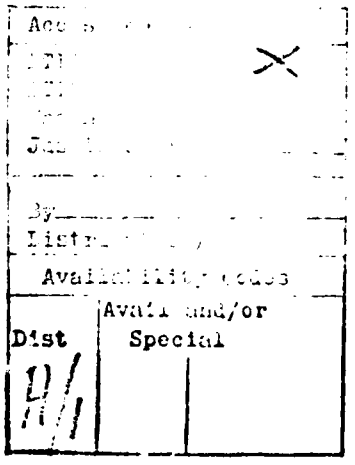
Ada in Mission Critical System Acquisition: A Guidebook

Dr. Siba N. Mohanty

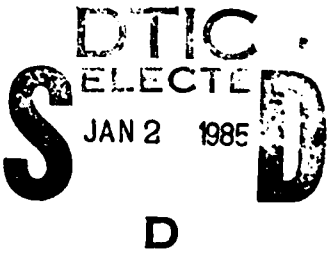
September 1984

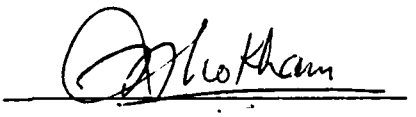
MTR-84W00189

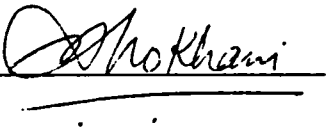
SPONSOR:
The Office of the Deputy Under Secretary of Defense (R&AT)
Ada Joint Program Office
CONTRACT NO.:
F19628-84-C-0001



The MITRE Corporation
MITRE C³I Division
Washington C³I Operations
1820 Dolley Madison Boulevard
McLean, Virginia 22102



Department Approval: 

MITRE Project Approval: 

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER: MTR-84W00189

2 GOVT ACCESSION NO.: AH9277

3 RECIPIENT'S CATALOG NUMBER:

4 TITLE (and Subtitle): Ada in Mission Critical Systems Acquisition: A Guidebook

5 TYPE OF REPORT & PERIOD COVERED:

6 PERFORMING ORG. REPORT NUMBER:

7 AUTHOR(s): Dr. Siba N. Mohanty

8 CONTRACT OR GRANT NUMBER(s): F19628-84-C-0001

9 PERFORMING ORGANIZATION NAME AND ADDRESS:
The MITRE Corporation
MITRE C3I Division
Washington C3I Operations
1802 Dolly Madison Blvd.
McLean, VA 22102

10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11 CONTROLLING OFFICE NAME AND ADDRESS:
Ada Joint Program Office
Room 3D-139
The Pentagon
Washington, DC 20301-3081

12 REPORT DATE: September 1984

13 NUMBER OF PAGES:

14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15 SECURITY CLASS (of this report): Unclassified

15a DECLASSIFICATION/DOWNGRADING SCHEDULE:

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unclassified

18 SUPPLEMENTARY NOTES

Final Version

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Acquisitions, Ada Programming Support Environment, Mission Critical Systems, Ada Programming Design Language, Life Cycle Software Support.

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report discusses several issues that influence the successful acquisition of major mission-critical system. These issues range from the role of software in the system acquisition process, and the state of software engineering practice in industry, to Ada compiler validation, Ada run-time environment, Ada programming environment, Ada resue, and Ada productivity. A project manager must deal with each of these issues during the course of his procurement and must devise plans to identify them and eliminate them in order to make his program a successful one.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

This report discusses several issues that influence the successful acquisition of major mission-critical systems. These issues range from the role of software in the system acquisition process, and the state of software engineering practice in industry, to Ada compiler validation, Ada run-time environment, Ada programming environment, Ada reuse, and Ada productivity. A project manager must deal with each of these issues during the course of his procurement and must devise plans to identify them and eliminate them in order to make his program a successful one.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

1.1 Background

Mission-critical computer systems are those systems--programs and data bases--that are developed as an integral part of defense system development activities. Mission-critical systems consist of command, control, communications, and intelligence (C^3I) systems and weapons systems. C^3I systems assist decision makers in gathering and evaluating intelligence information on enemy forces; enable commanders to assess the status and disposition of friendly forces; provide communication links over which orders and information can be transmitted; and give civilian and military leaders a means of redirecting their forces in response to changing conditions. C^3I systems serve five principal mission areas: nuclear force management; defense-wide information and communications; electronic warfare; command, control and communications (C^3) counter measures; and defense intelligence. Weapons systems are used in target acquisition, target selection, target designation, and in-flight control and delivery of weapons to designated targets.

Embedded computers are those systems that are integral to a weapons system or a C^3I system that performs complex, time-critical functions. Embedded systems are required to perform parallel processing, real-time control, and exception handling, and have unique input/output control requirements. Embedded systems are also used to perform real-time process control and transaction processing functions. In addition to embedded systems, the other major component of a C^3I system is data processing in a distributed environment.

Mission-critical systems in the military tend to be large (500,000 to 1,000,000 lines of code), be long lived (10 to 15 years of operational life), have real-time processing requirements, have severe physical constraints (due to lower processing speed and lim-

ited memory capacity of target computers), tend to be extremely complex and are expected to operate flawlessly under all operational conditions.

Computer software consumes the biggest portion of the embedded systems' budget. During 1973, the Department of Defense (DoD) spent $3 billion in procuring embedded software, which was 56 percent of the total embedded systems' budget. Estimates show that the cost of embedded software is expected to grow to $32 billion by 1990.(1) This spiralling growth in embedded software cost is of concern to DoD planners and the U.S. Congress. Embedded software problems are due to improper use of existing software technology (see the Appendix) as well as to a lack of a standard programming language for all mission-critical systems.

The potential benefits of a standard programming language can be illustrated from a study of an Army software system.(2) The system's 91 major software components were developed in 43 different programming languages on 58 different computer systems built by 29 manufacturers. Each component system has its own unique support software (e.g., operating system and utilities). If the Army had made use of a standard programming language to develop all the software components, significant cost savings, both for the development and maintenance, would have been possible.

For embedded defense systems, the operational environment (e.g., threat scenario) must change rapidly. To accommodate this changing operational environment, the software is constantly maintained (e.g., enhanced). The maintenance cost of an embedded defense system is estimated to be 75 to 80 percent of the total life cycle cost of the system. Adoption of a single, standard programming language will facilitate the use of one support software system, reduce training effort, promote personnel commonality, and offer substantial savings in both development and maintenance of embedded defense systems.

Also, the availability of large numbers of programmers to produce the required software in the future is a major concern. A common higher-order programming language and proper use of software technology will improve programmer productivity and software quality, and will result in lower overall cost for mission-critical defense systems. These factors motivated the DoD to develop a higher-order programming language, Ada (Registered Trademark of DoD, Ada Joint Program Office (AJPO)) and a software tool-set, Ada Programming Support Environment (APSE), to increase programmer productivity and improve software quality of mission-critical systems.

1.2 Purpose

The purpose of this report is threefold: (1) to explore the current approaches to acquiring mission-critical defense systems, (2) to explore how the acquisition process can be improved to provide greater visibility to software components of the system during the acquisition process, and (3) to explore the benefits of using Ada in the development of mission critical defense systems.

1.3 Outline of Report

Section 2.0 discusses the current approach to acquiring mission-critical computer systems using the milestone concept. The current practices of software engineering in industry, and how such practice (or lack thereof) influences system reliability and programmer productivity, is discussed in Section 3.0. Ada-based technology and experience are discussed in Sections 4.0 and 5.0, respectively. Finally, risk areas, and recommended ways to overcome these, are discussed in Section 6.0. A discussion of software engineering practice in industry is given in the Appendix.

2.0 SYSTEM ACQUISITION MANAGEMENT

Acquisition management involves understanding and applying methods, procedures, and tools for procuring a computer system to satisfy a well-defined mission need. Major mission-critical defense systems are procured under DoD Directive (DoDD) 5000.1,[3] whereas major Automated Information Systems are procured under DoDD 7920.1.[4] The current approach to acquiring mission-critical computing systems is based on the milestone concept (see Figure 2-1). There are four major milestones during the acquisition of a major system. These milestones occur at the end of well defined phases: Mission Analysis, Concept Exploration, Concept Validation [illegible] Scale Development. A decision for full-scale production is made at the end of the full-scale development phase. At each milestone the mission need, project progress to-date, costs, schedule, and risks are reviewed to determine if the project should be advanced to the next phase. The various acquisition phases and associated milestones are discussed below (see Figure 2-2).

2.1 Mission Analysis and Milestone 0

The procuring agency component(s) must undertake a continuing analysis for current and forecasted mission capabilities, technological opportunities, overall priorities, and resources that are involved in the acquisition of the proposed capabilities. When the analysis identifies a deficiency in existing agency capabilities or an opportunity to establish new capabilities in response to a technologically feasible opportunity, these will be formally set forth in a Mission Elements Needs Statement (MENS). Mission need must include mission purpose, capability, agency components involved, time constraints, value or worth of meeting the need, relative priority, and operating constraints. The statement is not to be expressed in terms of equipment or other means that might satisfy the need. The mission need statement is submitted to the agency head for approval at

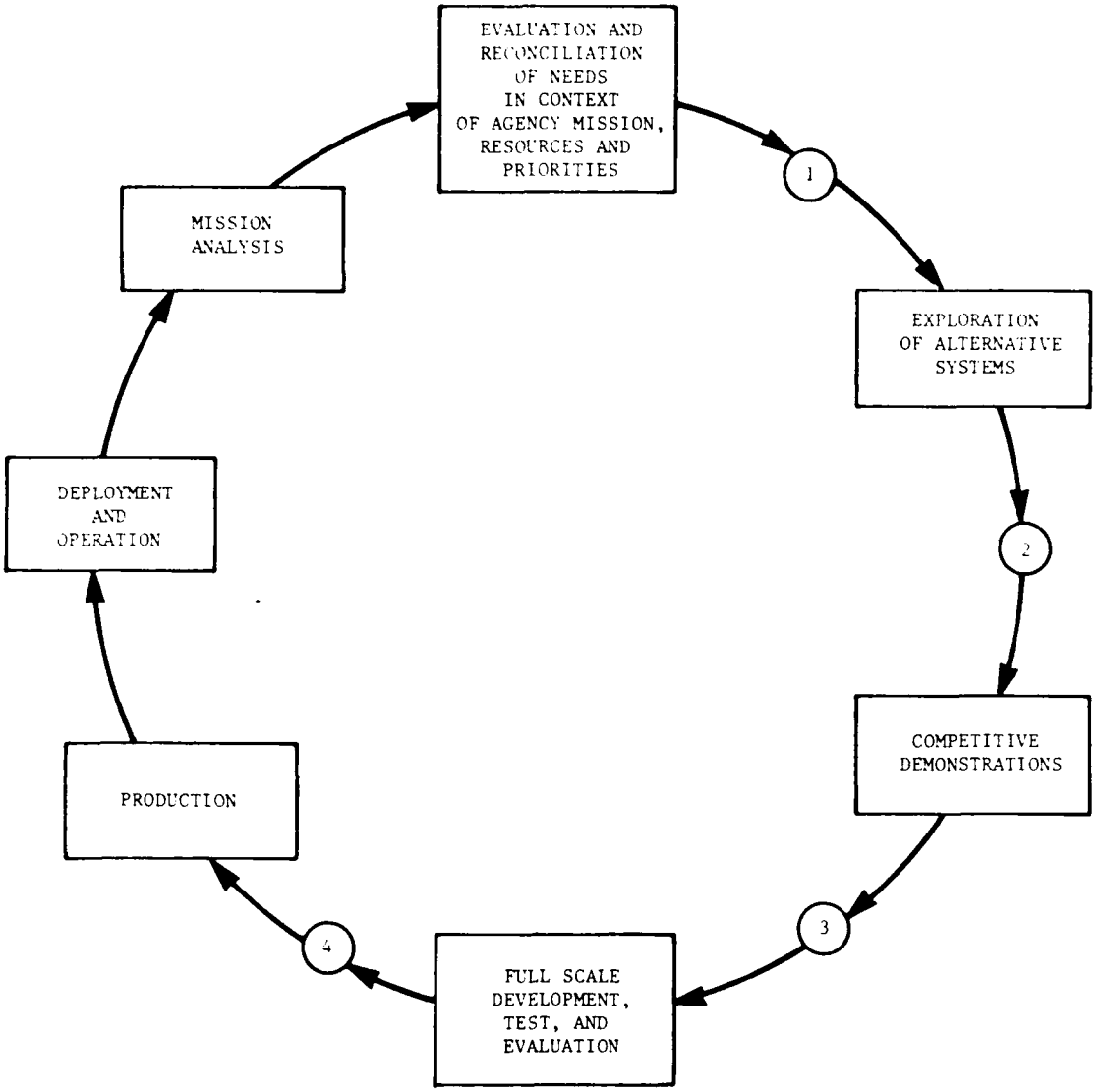


FIGURE 2-1
MAJOR SYSTEM ACQUISITION CYCLE

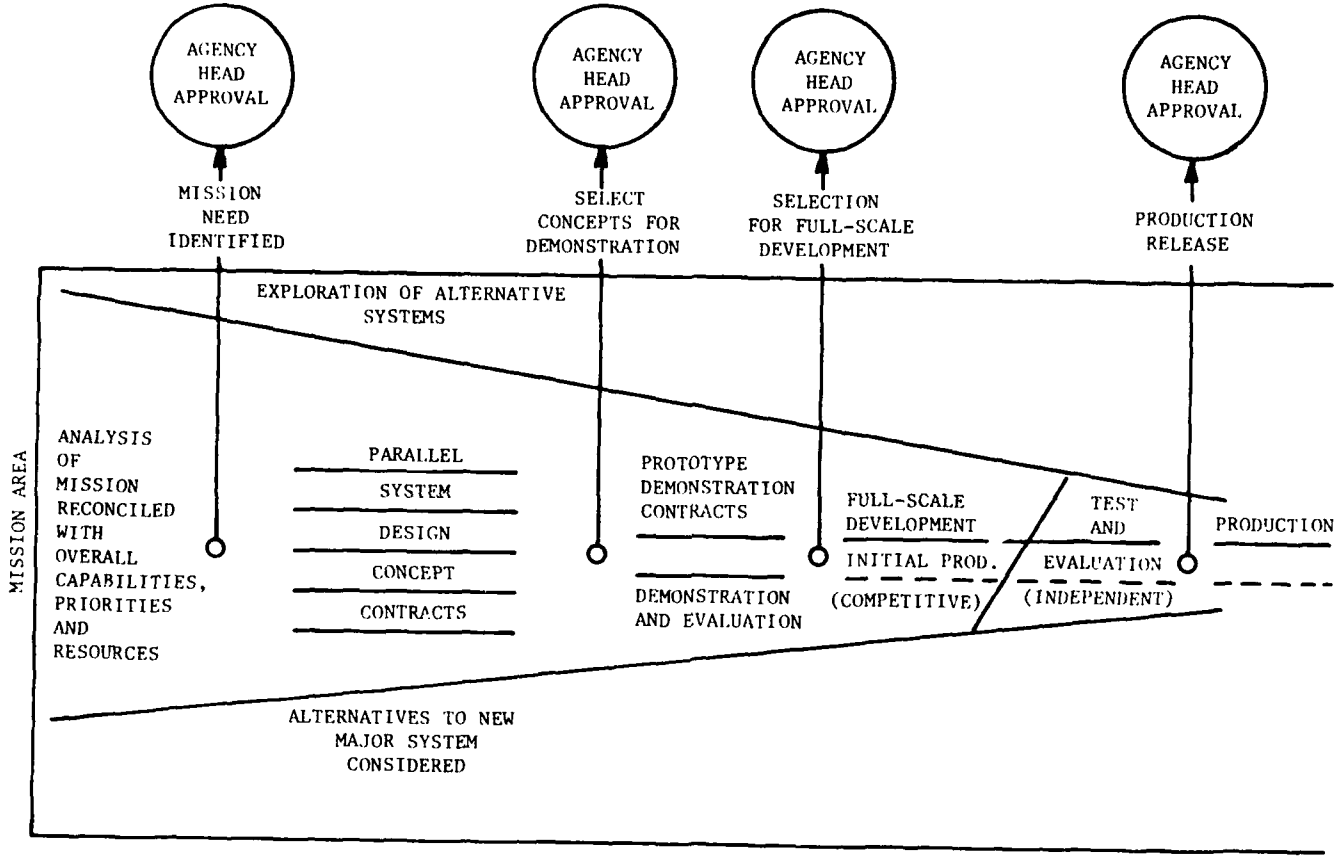


FIGURE 2-2
MAJOR SYSTEM ACQUISITION PROCESS

Milestone 0. If approved, the agency component(s) can move forward with the confidence of having a need recognized. Then, the exploration of alternative concepts begins to satisfy the mission requirements.

2.2 Exploration of Alternative Concepts and Milestone I

Approval of the mission need starts the major system acquisition process by granting authority to explore alternative system concepts (see Figure 2-3). This initial approval and the establishment of a system acquisition program do not automatically mean that a new major system will eventually be acquired. With an approved need, designated agency component(s) must continue to analyze other optional means of satisfying the needs in parallel with the exploration of alternative system concepts that must, as development proceeds, prove unacceptable. An evaluation of the options, including the alternative system design concepts at Milestone I, provides the basis for subsequent key decisions in the major system acquisition process. Prior to Milestone I, the designated program manager must explore system acquisition strategies and develop a system acquisition plan to implement the strategies that span the entire acquisition process. At Milestone I, the project undergoes review to evaluate the alternate concepts out of which one (or more) concept(s) is(are) selected for demonstration and validation through prototype development.

2.3 Concept Validation and Milestone II

Concept Validation (or, Advanced Development) intends to demonstrate that one of the alternative concepts explored thus far is valid and is capable of satisfying the mission need; that the system will perform in an operational environment; and that the system can be the basis for full-scale development of the system. Such concept validation normally involves some type of prototype development--this

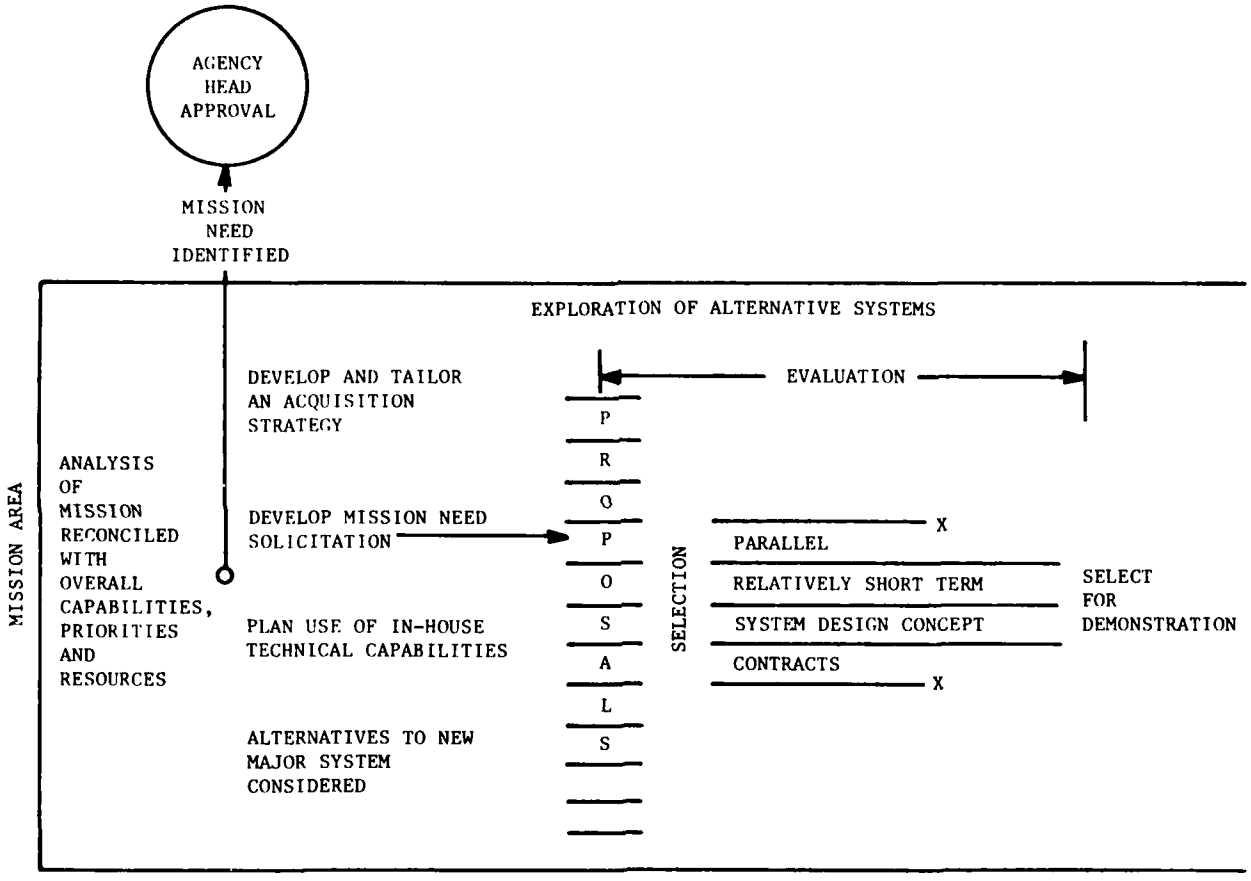


FIGURE 2-3
EXPLORATION OF ALTERNATIVE SYSTEMS

must range from a principal end item or critical subsystem to a limited and less-than-complete development model. Milestone II occurs after concept validation and before the full-scale development phase. At Milestone II, the results from demonstration and validation of alternate concepts are evaluated, and one of these prototypes is selected for full-scale development.

2.4 Full-Scale Development and Milestone III

Full-Scale Development (or, Full-Scale Engineering Development) and the initial production decision is made on the basis of the following:

a. Essential system concept projected performance measured against mission need and program objectives

b. An evaluation of remaining risks and potential resolutions

c. An evaluation of estimated acquisitions and ownership costs

d. Such factors as the contractor's demonstrated management, financial, and technical capabilities to meet program objectives.

Initial production units are to be tested and evaluated in an environment that ensures effective performance in expected operational conditions. Normally, the testing is to be done independent of the agency's development and user organizations. Exceptions to independent testing must be authorized by an agency head under such circumstances as physical or financial impracticality or in case of extreme urgency. Milestone III occurs after the full-scale development phase. This is the final milestone at which a decision is made regarding the full-scale production of the system.

In addition to these four major milestones, interim project reviews are held to address and resolve technical and management issues that must surface during the development effort.

3.0 MISSION CRITICAL SOFTWARE ACQUISITION

3.1 Problem Areas

For embedded systems, the existing acquisition process treats software as an integral part of the hardware. That is, even if the embedded software constitutes a major part of the prototype system and often is the high-risk item, it is never treated as a separate entity for procurement purposes. Under current acquisition practices, a prototype system is made available at Milestone II. In order for the prototype to function correctly, all of the software components of the prototype system must be available at Milestone II. That is, the software must be produced and acceptance tested during prototype development of the system. Even though both hardware and software have the same phases in the development cycle (requirements analysis, specification, design, development, and test and integration), the time lines for the software and hardware phases do not match. Most major life-cycle events (e.g., design) occur earlier for software than similar events for hardware. In other words, by the time the system reaches Milestone II, the software has undergone a complete development cycle, whereas the hardware is at the preliminary design stage. During full-scale development of the system, the hardware is changed to improve system performance or to accommodate new requirements. Often such changes to the prototype hardware imply significant changes to the software. Since the software has undergone acceptance testing, any changes to software at this stage tend to be expensive. Even then, in most cases, any performance improvement in the system prior to Milestone III can only be achieved by changing the software. The realization that the software must undergo significant changes necessitates that the software be properly designed, developed, and documented so as to be maintainable.

The differences in time lines for major reviews for hardware and software can be accommodated by requiring that formal reviews for software must precede the similar reviews for hardware. That is, Requirements Review, Design Review, Preliminary Design Review, Critical Design Review, and Test Readiness Review for software must precede the similar events for hardware. In fact, for software, all of these reviews must be scheduled during the Concept Validation phase of the acquisition process and prior to the Milestone II review. This will facilitate orderly development of software and will enable the contractor to maintain (i.e., enhance) the software during the full-scale development phase.

3.2 Ada in Mission Critical Acquisitions

DoDD 5000.29 emphasizes the procurement of reliable software under various DoD and Services Acquisition regulations.[5] The three military departments of DoD use separate regulations to enforce this approach (i.e., Air Force Regulation (AFR) 800-14;[6] Army Regulation (AR) 70-XX;[7] and Secretary of the Navy Instruction (SECNAVINST) 5200.32.[8]). To inject the new technologies into the acquisition process, DoDD 5000.31[9] requires that the Ada programming language become the single, common computer programming language for Defense mission-critical applications. Effective 1 January 1984 for programs entering Advanced Development and 1 July 1984 for programs entering Full-Scale Engineering Development, Ada shall be the programming language. This directive need not be applied retroactively to DoD systems for which formal language commitments have been made in compliance with DoDD 5000.31 before the effective date of this revision.

AR 70-XX, for example, requires that the Computer Resources Management Plan (CRMP) must spell out the requirements of using Ada as a Program Design Language (PDL) and an implementation language. The following are acceptable plans for using Ada as the implementation language (stated in order of preference):[7]

(1) The Army's Ada Language System (ALS) is used for the development and maintenance of mission-critical software. This is the long-term Army objective. However, the ALS will be available for initial Army use in January 1985 and is targeted to only a small number of machines.

(2) A validated Ada compiler other than the ALS is used and all Ada code developed for the Army is transitioned to the ALS before Department of the Army Development and Readiness Command (DARCOM) Life Cycle Software Support (LCSS) accepts it for maintenance. The project manager will fund the development of appropriate code generators and hosts for the ALS if code generators are not in progress for his hardware. Current targets are the VAX 11/780 with ALS/VMS, the Nebula military standard (MIL-STD)-1862B, the Intel 8086 microprocessor, and the Navy's AN/UYK-44.

(3) If an unvalidated Ada compiler is used to start development, the project manager will take the following steps, in addition to the requirements in (2) above, to ensure that mission-critical software using variants of the Ada languages are not accepted by the Army.

(A) For the purpose of source selection, unvalidated Ada is not Ada. No additional credit will be given to an offerer proposing to use an Ada compiler that is currently not validated.

(B) For any software developed in Ada, the contract shall stipulate that the government will not accept the software until it has been compiled with a validated Ada compiler.

The DoD and Army directives, it appears, apply to mission critical procurements that are starting anew or are stand alone. However, major mission-critical procurements must make use of existing systems as critical components, and existing systems must be the starting point for a major future system.

Existing systems often have significant amounts of operational software written in assembly language or another high-level language (e.g., FORTRAN, JOVIAL, or CMS-2). Consequently, if a mission-critical procurement uses an existing system and associated software, the project manager either continues using the old programming

language for additional program implementation, or develops and implements a transition plan for converting the old programs into Ada and then continues using Ada as required by the above directives. Such a choice can be made only after careful evaluation of tne alternatives in terms of cost-benefits and risks associated with each of them.

A similar situation exists for systems for which language commitments have been made earlier. Even though the use of a non-Ada language must be sufficient for these projects, an analysis should be made as to whether this system will be a starting point for major mission-critical systems in the future. If the system is likely to be a starting point for a major acquisition, then the Program Manager must plan to transition the non-Ada software into Ada at the conclusion of the project.

3.3 Ada Transition Plan

As discussed earlier, a project might have a significant amount of software written in a language other than Ada that needs to be reused. Also, the current project might be a starting point for a major future system. For commonality, the Program Manager must develop and implement a Transition Plan so as to convert tne non-Ada software into Ada. A brief outline of such a plan is given below. As a minimum, a Transition Plan should:

a. Define the scope of the (conversion) effort

b. Describe the host and target hardware

c. Describe the existing and future software architectures

d. Indicate explicit task assignments and define responsibilities of the transition group

e. Allocate and schedule resources (manpower, computers, etc.)

f. Document the technical approach, key assumptions, and constraints underlying the existing system (e.g., hardware and software), and any deviations for the future system

g. Define appropriate interfaces between the hardware and software configuration items and project teams

h. Document standards and practices to be used in the transition effort

i. Specify a definite completion criterion to be used (e.g., if the target programming language is Ada, then the Ada code should be compilable by a validated Ada compiler; if the target code is in assembly language, then it can be invoked and executed by an Ada program)

j. Provide adequate testing requirements prior to acceptance.

Several prototype translators are available to translate existing FORTRAN and COBOL programs into Ada programs.[10] The approach is to use a COBOL or FORTRAN analysis tool and produce DIANA (Description Intermediate Attributed Notation for Ada) intermediate form representation from which the Ada code can be decompiled. This approach will try to translate 100 percent of the source code automatically without human intervention. The approach is being tried in the Worldwide Military Command and Control System (WWMCCS) Information System (WIS) project.[10]

4.0 Ada BASED TECHNOLOGY

4.1 Ada Language

Since no single DoD standard programming language satisfied all the requirements of embedded software, DoD commissioned the development of a new programming language.[11] This extensive effort resulted in a new programming language, Ada. The Ada specification effort began in 1976, and Ada became a standard programming language in 1983.[12] Ada language features are briefly discussed below; readers interested in more detailed technical information on Ada must refer to other sources.[13-15]

Ada was designed for ease of program development and maintenance.[16] To achieve these objectives, Ada makes extensive use of the principles of data abstraction and information hiding. Abstraction refers to the ability to differentiate between the functional characteristics of an object from the implementation of these characteristics in the form of a program. Information hiding refers to the ability to hide object-unique information from other objects. Ada makes use of these concepts extensively in the design of subprograms, packages, tasks, and generics. A program in Ada is a collection of functional components and data objects that interact serially or in parallel. These functional components can be viewed as subprograms, each subprogram having a specification part and a body. Ada, by virtue of its design, allows the user to define subprograms and subprogram stubs. During top-down design, subprogram stubs with subprogram specification only are used to design and test the system in a top-down manner. The stub bodies are implemented after the design is complete and is verified to be correct.

A package or software module or program-unit in Ada is the highest level of abstraction. A package is a collection of groups of related types of objects and subprograms. A package has two parts:

the specification and the body. The package specification is an abstract interface to other packages, and its type can be made visible or kept private. The visible part of the package specification describes the functional characteristics and logical interfaces of the package. The private part specifies the physical interfaces that a compiler needs to know during the compilation of the package specification. Information in the private part cannot be used by the programmer in any way. The package body implements the specification of the package. The specification and the body are two separate physical program units, which are compiled and catalogued separately. The packages facilitate maintenance in the following manner:

a. Modification to a package body will not result in modification to or recompilation of source programs using these packages.

b. Modification to the private part of the specification will not result in modification to source programs using these packages, but must require recompilation of the source programs.

c. Modification to the visible part of the package specification will result in changing the source programs and will require recompilation.

Ada package specification and package body can be compiled separately from each other. This is possible due to strong data typing of the package interfaces and variables and due to the provision of a compilation data base. The compilation data base stores the compilation information that is used in compiling any program unit. The compilation is context dependent; that is, a package body cannot be compiled if the respective package specification has not been compiled.

Ada facilitates real-time programming for embedded systems through the provision of tasking. Tasks are used to execute concurrent activities. As in packages, a task has the specification

and a body. The task specification consists of a declaration for the task entry points. The task body implements the task specification. The tasks communicate through "rendezvous." That is, when task A calls task B, a rendezvous is said to occur when execution of an ENTRY from task A and an ACCEPT from task B coincides. The data transfer occurs during task rendezvous.

Generic program units in Ada permit the system designer and programmer to create subprograms and packages that provide computing with abstract data types. That is, if a generic Ada sort routine is available for sorting integer arrays, it can be adapted to sort an array of names by changing the data types at compile time. Generic units define templates of program units that are written once and then tailored to a particular need during translation time. For example, generic units can be developed for vector operation or matrix operation, and the precision of these operations can be specified as a parameter during compile time.

4.1.1 Ada Compilers

At present, several software companies are developing Ada compilers that are being validated by the AJPO. As of this writing, Data General/Rolm, Telesoft, and Western Digital/Gensoft Ada compilers and New York University (NYU)/Ed interpreters have been validated by AJPO. Two significant characteristics of a compiler are the compilation speed and the run-time support environment. Ada programs are likely to be larger than equivalent programs in other languages. Ada is large, and extensive type checking is performed at compile time. Hence, compilation speed of Ada compilers is likely to be slower than the compilers of other languages. Ada compilers often require 0.5 to 1 megabytes of main memory. Due to complex data structures, Ada compilers might require 1 to 2 megabytes of main memory for compilation purposes. In addition, for efficient operation, large amounts of secondary memory will be required.

Consequently, the Ada development environment should consist of large amounts of main memory and secondary memory for efficient compilation of Ada programs. As the compilers mature, the speed of Ada compilers will increase. In the future, the compilation speed of Ada compilers must not be a major risk area in any project.

In the near term, compiler validation issues, however, will continue to be a major issue for most projects. This is due to the fact that DoD and Service directives require that, prior to final delivery, Ada software be compiled by using a validated Ada compiler.

AJPO requires that a validated compiler be revalidated once a year. That requirement is likely to be changed to once in six months in the future. The revalidation is required to (1) identify any changes to a validated compiler; (2) identify and eliminate errors in the validation suite; and (3) develop a new validation suite for previously untested language features. In other words, a validated compiler must be invalidated for one of three reasons, two of which must be beyond the control of the compiler developer. In order to guard against the possibility that a validated production compiler is invalidated in the course of the project, a project manager must plan to acquire more than one production compiler or must restrict the use of certain language features for his project.

4.2 Ada Target Environment Characteristics

The Ada target computing environment, the embedded environment, is likely to be no more than a bare machine without a run-time support system. Run-time support consists of utility routines that are required for execution of the object code in the target machine. Since the target machines do not have any run-time support, any run-time support of this kind in the development environment must be transported to the embedded environment.

Many existing target computers were not designed to execute Ada efficiently. In other words, the instruction set of many target computers must not be suitable to generate efficient machine code for Ada programs. But, in applications with hard real-time constraints, efficiency of the target code is a necessity. In order to achieve this efficiency, many code generators must resort to short cuts in implementing the code generators.(17) These short-cuts or optimizations will generate target code that must be efficient but must not have one-to-one correspondence between the source and the target code. Consequently, if one uses an Ada compiler with an optimized code generator, then in case of errors in target code, debugging will be very difficult. This fact must impede maintainability of these Ada programs. A project manager must weigh these risks, and if the performance requirements of the target code for certain real-time functions are not achievable by straight compilation, he must utilize a language other than Ada for implementing these real-time functions.

4.3 Ada Programming Support Environments

Two parallel efforts are now underway to provide APSEs for Ada programming efforts. ALS, sponsored by the U. S. Army and developed by SofTech,(18) is expected to provide an Ada Compiler, system utilities, and facilities for large-scale program development (e.g., library support and version control, etc.). It will be hosted in a DEC VAX 11/780 computer. Ada Integrated Environment (AIE), sponsored by the U. S. Air Force and developed by Intermetrics,(19) is expected to provide an Ada Compiler, system utilities, and facilities for large-scale program development. AIE will be hosted in an IBM 370 computer. The ALS is planned to be delivered soon, but the delivery of AIE is indefinitely delayed. The tools in ALS and AIE are integrated, that is, the tools interact with a common data base that characterizes the program and its associated environment. The WIS project, on the other hand, is procuring an Ada Programming Environ-

ment, CONAN, whose tools are not integrated.[10] To facilitate movement of programming environments among various operating systems, a standard Common APSE Interface Specification (CAIS) is being developed.[20]

These programmming environments are extremely beneficial to develop high quality Ada programs. Each programming environment includes a unique Ada compiler that must or must not produce efficient code for the target computer used in the project. If such is the case, the project must substitute a compiler that can produce appropriate object code for the target computer. If a substitute compiler cannot be used in the selected programming environment, the project must not utilize the Ada programming environment to develop the source code, but should utilize stand alone compilers and programming support tools in the project.

5.0 Ada EXPERIENCE

5.1 Ada as a Program Design Language

Since changes to the embedded software are inevitable, the change process should be easy and reliable. Software design is the most important element that influences later maintainability. Consequently, to develop maintainable software, its design should be correct and complete. Since the Ada language has the features to develop good designs, Ada programs are also highly maintainable.

Computer program design is a complex task that often requires the assimilation of large quantities of information and the development of solutions to satisfy many conflicting requirements. Various methodologies, such as the Jackson Approach,[21] the Warnier-Orr Approach,[22] Top-down Design, and Structured Design,[23-26] can be used for software design activity. Ada facilitates the software design process by providing abstraction and information-hiding facilities. These features are easily exploited during high-level design, preliminary design, and detail design. During preliminary design, a package with appropriate interface specification is provided. Once the preliminary design is verified, the detailed design commences during which the body of the package is implemented.

A PDL is used to document a program design in lieu of using flowcharts. PDL is a pseudocode that describes the control structure and general organization of the program. A free-form PDL contains a minimal number of syntax and semantic rules. A design language has three principal uses:[27]

a. As a medium for the formulation of the design

b. As a means to communicate the design

c. As a medium to formally document the design.

PDL normally provides (1) indenting and labelling facilities; and (2) control structures consisting of selection (IF-THEN-ELSE), iteration (WHILE), and sequence and module invocation (CALL). The use of Ada as a PDL has many advantages:[28]

a. It allows for syntax checking by the Ada compiler.

b. It is a convenient method of expressing interface specification between computer program components. The specification can be checked for correctness, consistency, and traceability.

c. Ada explicitly supports software engineering principles, i.e., a designer well versed in software engineering can create proper designs.

d. Use of Ada as a PDL will encourage the contractor and program management to learn the language, which, in turn, will facilitate the introduction of a new language into the project and accelerate compiler maturation.

As mentioned before, a PDL is a means to document a design at the preliminary and detail design stages of system development. Preliminary guidelines to use Ada as a design language have been developed by an Institute of Electrical and Electronics Engineers' (IEEE) software engineering standards working group.[29] Successful use of PDL depends on a successful design. The experience of a group of engineers at General Electric (GE) demonstrates this.[30]

The GE team undertook the task of redesigning and implementing an existing FORTRAN and Assembly-based system in Ada. The subsystems to be redesigned were (1) an analysis of multiple beam antennas; and (2) a graphics package to display the results of the analysis. The effort resulted in a new design that was implemented in Ada. The observations of the team are as follows:

a. "...The word "Ada" has been used to mean both the language itself and various design concepts supported by the language.... unless explicitly trained...programmers will utilize Ada in a manner similar to FORTRAN and Pascal and

will not make effective use of structuring and information-hiding facilities that Ada provides."

b. "...A comparision of two designs illustrates that a team's background and design philosophy can have a more profound effect on the ultimate design than the choice of the language. That Ada supports information-hiding is no guarantee that a development team will exploit this technique. If one believes that information-hiding is desirable, one must address many other issues in addition to the choice of language. These include the background and experience of the team, the training, and the requirements document."

c. "...Most short Ada courses tend to teach the language itself, instead of presenting an overall design philosophy that can be expressed through Ada."

The foregoing observations illustrate that proper software design is very important for the development of reliable and maintainable software. A project must have to undertake many steps to ensure that a good design exists prior to expressing the design in a PDL. One approach is to develop a Module Design Guide (MDG), which can be used to design the software modules for the project.(31) The idea of MDG has been developed by Parnas and others at the Naval Research Laboratory in response to a, "...growing gap between software engineering principles being advocated in major conferences and the practice of software engineering at many industrial and govermental laboratories... The ideas appeared to be easier to write about than to use."(31)

The MDG is concerned with three structures: (1) the module structure and the assumptions made about various modules; (2) the "uses" structure--the relation "requires the use of"; (3) the process structure--the run-time activities involving modules. The module structure is based on the principle of information-hiding; that is, module details that are likely to change independently should be the secrets of separate modules; the only information that appears in

the module interfaces is that which is unlikely to change. The MDG is intended to satisfy the following goals:

a. One should be able to understand the module responsibility without understanding the internal design of the module.

b. One should be able to identify a group of relevant modules from those that are irrelevant to suit one's need.

c. The number of branches in and out of a module--number of interfaces, should be sufficiently small so as to track the affected modules in the event of a change to a given module.

A module guide can be developed by using the software engineering principles so as to achieve the goals described above. Readers familiar with Ada syntax can see the close relationship between the module guide and the Ada language features. Thus, software modules designed using the MDG can be easily implemented in Ada PDL.

In using Ada in his project, a project manager must be concerned with two distinct activities: (1) a proper module design; and (2) expressing the design using suitable features of Ada. It is suggested that an MDG be used for proper design of software modules and a PDL guide (e.g., an improved version of IEEE PDL guidebook) be used to express those designs. Then, and only then, is the resulting design likely to impact the development and maintenance phases of the system life-cycle and is likely to result in reliable and maintainable Ada programs.

5.2 Distributed Computing in Ada

Modern C^3I systems are a collection of physically dispersed data processing sites with associated data bases. These systems must include embedded, as well as distributed, systems. Effective C^3 depends on coordinated activities among all dispersed sites. To achieve effective C^3, significant distributed computing is required. In the following sections, several issues relevant to distributed computing will be discussed.

A distributed computing environment is defined as a collection of distributed processing centers and data bases, physically and logically interconnected by communication links for the purpose of executing applications programs in a coordinated manner.(32) In distributed C^3 systems, processes communicate through message passing. An interconnection network routes messages to appropriate processes and all interprocess interaction occurs at the network level. Communications primitives that facilitate applications level interaction can be classified into two types of communication, as follows:(32)

a. Synchronous communication

   1. Synchronous communication with remote invocation (exactly once, or at least once)

   2. Synchronous communications without remote invocation.

b. Asynchronous communication.

For synchronous communication, a sending process sends a request to a receiving process indicating its desire to send. The receiving process accepts the request and indicates its willingness to accept the data by returning an acknowledgement message. When both the sending and receiving processes are at a predetermined state (e.g., synchronized), the data transfer between processes occurs. For widely dispersed processing centers, the time delay for message transfer must be unduly large, and thus must rule out the possibility of concurrent processing of processes. That is, the sending process must remain inactive until the receiving process has acknowledged and accepted the data from the sending process. Asynchronous processing does not require that the processes be synchronized for data transfer. That is, in asynchronous communication, data transfer occurs without acknowledgement by the receiving process. Asynchronous communication takes place through the provision of data buffers at the receiving end. Asynchronous communication also facilitates

concurrent processing of processes since the sender does not have to wait for the receiver.

Synchronous communication, with or without remote invocation, can be performed using Ada Tasks. In implementing the synchronous communication, the programmer uses a DISCONNECT statement to delay the acknowledge statement. Placement of the DISCONNECT statement is an indicator of the level of concurrency desired. The placement of this statement has to be carefully calculated by the applications programmer and is often tricky. Asynchronous communication using a data buffer, on the other hand, does not have any such requirements. Asynchronous communication requires three processes: a sender, a receiver, and a bounded-buffer. An example of such a model implemented in Ada is given below.(32) In this example, the buffer has 10 slots that can be filled by the sender and emptied by the receiver. All three processes execute asynchronously. If the buffer is full, the sender cannot fill the buffer. Similarly, if the buffer is empty, the receiver cannot empty the buffer.

```
TASK Buffer IS
     ENTRY send (p1:IN portion);
     ENTRY receive (p2:OUT portion);
END;

TASK BODY Buffer IS
     buf: array (0..9) OF portion;
     inp:  integer:=0;
     outp: integer:=0;
BEGIN
     LOOP
         SELECT -- nondeterministically
         WHEN inp-outp <10=> --not full
            ACCEPT send (p1: IN portion) DO
                   buf (inp  mod10):=p1;
            END; --disconnect
            inp:=inp+1;
         OR
           WHEN inp>outp => --not empty
              ACCEPT receive (p2:OUT portion) DO;
                    p2:=buf(outp mod10);
```

```
            END; -- disconnect
            outp:=outp+1;
        END SELECT;
    END LOOP;
END BUFFER;
```

```
TASK BODY sender IS
   .
   .
  LOOP
    buffer.receive (m);
   .
   .
  END LOOP;
END sender;
```

```
TASK BODY receiver IS
      .
      .
    LOOP
      buffer.send (m);
      .
      .
  END LOOP;
END receiver;
```

The processes (tasks) communicate when execution of ACCEPT of one of the process and the ENTRY of the other process coincide (e.g., rendezvous) during execution.

Data base access and update is the next most important thing in distributed processing environments. This data base support can be provided by designing interfaces to existing data base systems or by specifying and designing an entirely new data base system that will fully exploit the capabilities of Ada.(10)

5.3 Use of Off-the-shelf Ada Packages

Packages are one of the more exciting concepts of Ada. The concept of the package must, one day, revolutionize the computing industry in the way programs are designed and developed. Future programmers will build programs out of existing packages (generic or library units) rather than build them from scratch. This is likely to improve the quality of future programs. Also, the cost of future systems using off-the-shelf Ada packages is likely be lower.

Experts estimate that eventually up to 90 percent of the program designers will use some portion of an existing Ada package from a catalog. Intellimac Corp. has developed a set of packages that can

be used along with a Telesoft Ada compiler to develop programs for commercial applications (e.g., Payroll).[33] These packages are as follows: Str_Handler, Dater, Terminal, Float_BCD, Num_Handler, Format_Chk, In_Out, and Screen_IO. It is expected that utility packages like these and mathematical application packages for matrix multiplication, multiple regression, etc. will be widely available to permit application programmers to use these packages instead of developing them.

Typical mission-critical Ada projects will consist of 100,000 to 2,000,000 machine instructions, which are equivalent to 20,000 to 500,000 lines of Ada source code.[34] Some of these future projects will use existing off-the-shelf Ada software extensively. Likely characteristics of such an Ada project are as follows:

a. Half of the programs will be off-the-shelf and the other half will be written specifically for the project.

b. Ninety percent of the new code will be in portable Ada. Use of machine language instruction and assembly subprograms will be discouraged.

Some potential problems in using existing off-the-shelf Ada packages in an Ada project are discussed below. In order to use an existing Ada package (generic or library unit), one has to first determine that there is a good match between the user specification and the specification of the existing package. If the package satisfies only a portion of the requirements, the package must be modified to "tailor" it for the application. Assume that a package specification is 100 lines of Ada code, and its body is 1000 lines long. If changes are required for the package, the maintenance must not be easy. Also, recompilation of the new package implies recompiling all the other packages used by the new package. These additional packages could mean an additional 2500 lines of Ada code. So, in using an

existing Ada package one is expected to recompile 3500 lines of Ada code on the average.[35, 36]

From this scenario, it can easily be infered that any large project trying to use off-the-shelf packages will have to implement a formalized maintenance procedure through rigorous configuration control. A good programming environment and formalized configuration management is a must for proper use of off-the-shelf packages.

5.4 Ada Productivity

Even though there is no definite measure of software productivity, lines of code/unit-time is still the most widely used measure in industry. Experience shows that, for complex real-time systems, productivity varies between 0.3 to 1.0 lines of delivered code/hour.[37, 38] So far, Ada has been used in several relatively small and less complex applications. The reported productivity figures vary between 5 to 8 lines of delivered code/hour.[39] Productivity figures for large, complex, real-time systems are likely to be lower than these figures, but they are likely to be higher than the 1 line/hour for non-Ada systems. The question is how should these higher productivity figures be considered in costing Ada-based systems?

To begin with, the productivity figures reported thus far are actually for the program development phase only. Since software development effort begins with requirements analysis and ends with test-integration, all these phases must be considered in cost estimation. For example, a large software system development expenditure is distributed as follows: 20 percent for requirements analysis and specification; 20 percent for design; 20 percent for development; and 40 percent for test-integration. Use of Ada is unlikely to influence and thereby reduce the requirements analysis and specification efforts. Even though the design will be influenced by Ada concepts,

there is no solid indication that the design effort will be reduced if Ada is used as a PDL. Consequently, for large software systems, the front end efforts for analysis, specification, and design must remain unchanged. If Ada is used for programming, the programming effort must be reduced by as much as a factor of five for small systems. But, for large systems, the actual reduction factor is yet to be determined. Additionally, there is no solid information as to whether use of Ada for programming will result in a reduced test-integration effort, but most experts will agree that a reduction in test-integration effort of Ada programs is likely. Consequently, from a development view point, the total reduction in development effort must not be significant and is likely to be on the order of 10 to 15 percent.

The cost of Ada programmers must also be considered in estimating the total system development cost. Since Ada is new, there are few proficient Ada programmers. The law of supply and demand dictates that these programmers will be higher priced than assembly language or FORTRAN programmers, at least in the near term. Hence, in the near future, even though a reduction in development effort is possible for Ada-based systems, the net total development cost must not be significantly different from a non-Ada system. On the other hand, the maintenance effort for Ada programs is likely to be significantly smaller than for non-Ada systems. Since maintenance consumes 75 to 80 percent of the total life-cycle costs, use of Ada is likely to result in reduced total life-cycle cost for the project. Consequently, a project manager planning to use Ada in a project must not realize significant gain in development cost but is likely to experience significant gain in the maintenance cost.

From this discussion, it is apparent that the Ada language is well suited to design and develop reliable and maintainable mission-critical software. At the same time, the discussion in the Appendix

indicates that the practice of software engineering is rare in the industry. Ada must be the catalyst that will accelerate the software engineering practice in industry and assist in the production of reliable software.

Additionally, experience shows that large systems benefit from management approaches and development methodologies.[40] Also, formalized quality assurance, configuration management, independent verification and validation, and test and evaluation are known to improve the quality of large mission critical systems.[41-43] Consequently, in addition to complying with Ada-related directives, a project manager must insist on a formalized management approach and development methodology as well as use of proven tools so as to minimize the risks associated with his procurements.

6.0 RECOMMENDATIONS

Several risk areas related to major system acquisition are discussed below. A program manager must analyze each of these risk areas in light of his particular project requirements and must develop plans and procedures to deal with them during the acquisition process.

a. The project manager must recognize the differences in time lines for development phases for hardware and embedded software and must schedule formal reviews accordingly.

b. If the project has a substantial amount of existing software, a cost-benefits analysis should be made prior to deciding whether or not to transition to Ada. A substantial amount of existing software in a language other than Ada must imply that the project should not use Ada for future program implementations. If the amount of existing code is not significant, a transition plan should be prepared and implemented for orderly transition to Ada.

c. If the current project is likely to be a starting point for a major future system, then, even if the project is targeted for implementation in a non-Ada language, the project manager must plan to convert the non-Ada software into Ada after program conclusion.

d. If the ALS (or a similar programming environment) is used for program development, then for some real-time problems, the target code must fail to meet the specified performance criteria. Consequently, the project manager must plan to develop critical real-time code segments in another DoD-approved, higher-order language that can be integrated with the Ada code.

e. If a code generator for a specific target computer does not exist, the project manager has to contract for the development of the appropriate code generator. As usual, there is a slim chance that the code generator will fail to produce object code that meets the performance requirements. To counter such a possibility, the project manager must require that the time-critical functions be developed in another programming language.

f. A validated compiler has to be used for the compilation of the final system prior to delivery. There is always a slight chance of an existing validated compiler getting invalidated due to new tests in the validation suite or due to discovery that some tests in the earlier validation suite are incorrect. Consequently, a contingency plan to use another compiler must be developed.

g. Ada as a PDL, if properly used, is likely to result in higher productivity and is likely to produce maintainable and reliable programs. Guidelines for proper use of a MDG and proper use of Ada as a PDL are a must for achieving these objectives.

h. Off-the-shelf packages can speed up development time of Ada-based systems. In order to achieve this benefit a coherent plan for Ada reusability, a program development environment and formalized configuration management procedures are required.

APPENDIX

SOFTWARE ENGINEERING PRACTICE IN INDUSTRY

Ada has been designed with software engineering in mind. The term software engineering is used to describe ways to develop, manage, and maintain software so that the resulting product is reliable, correct, efficient and flexible. The language designers have tried to implement data abstraction and information-hiding and have provided packages, tasks, generics, subprograms, stubs, abstract data typing, and pragmas, to name a few, that support the use of software engineering principles--system decomposition, stepwise refinement, and modularization--in system development. The current trend is to associate software engineering with Ada, so much so that many consider software engineering to be Ada's middle name (as in, Ada "Software Engineering" Augusta Byron, etc.). Success of Ada in building mission critical defense systems is closely tied with the proper practice of software engineering.

A recent survey [44] by a team of researchers from the University of Maryland addresses this issue. In the following paragraphs, sections of their paper are abstracted to give an overview of the use of software engineering in the data processing industry. From their survey, it is clear that the practice of software engineering in industry is, indeed, rare. Before Ada can be used successfully in developing reliable and maintainable software, awareness and practice of software engineering must be improved. Until then, mere use of Ada as a PDL or an implementation language must not yield the expected benefits.

The survey started in June of 1980 and continued through the summer of 1983, involving 30 organizations: 5 IBM divisions, 12 other

U. S. companies and 13 Japanese companies. The survey consisted of 2 small (less than 10 staff-months), 7 medium (10 to 100 staff-months), 11 large (100 to 1000 staff-months), and 9 very large (greater than 1000 staff-months) projects. Some of the findings are given below.

"...We found surprisingly little use of software engineering practices across all companies. No organization fully tries to use the available technology. Although some companies had stronger management practices than others, none used tools to support these practices in any significant way."

"...Tools are not widely used in the industry. Not too surprisingly, the use of tools varies inversely from how far the tools are from the code development phase. Tools are most frequently used during the code and unit test phase of software development (e.g.,compiler, code auditor, test coverage monitor, etc.). Tools are less frequently used in the adjacent phases of the software life-cycle--design and integration (e.g., PDL processors and source code control systems). Few requirements or maintenance tools are used."

"...At all places contacted, requirements were in natural language text. Some projects had requirements documents that could be processed by machine, but tool support was limited to screen-oriented text editors. No analysis tools (like Software Requirements Engineering Methodology (SREM) and Problem Statement Language (PSL)/Problem Statement Analyzer (PSA)) were used, except on toy projects."

"...Most designs are expressed in some form of PDL or pseudo code.... While the use of PDL seems to be accepted practice, its use is not particularly effective. For example, we consider the expansion of PDL to code a reasonable measure of the detail in a design. A PDL-to-source-code expansion ratios of 1:1 must indicate that the design has the same detail as its eventual code. With design and code

being separate tasks, this expansion indicates that the two concepts are not separated. The expansion ratio of PDL-to-source-code were 1:5-10 at one location, 1:3-10 at another, and 1:1.5(!!)-3 at a third."

"...All companies surveyed had a methodology manual; however, they were either out of date, or were just in the process of being updated. In this environment, Department of Defense MIL (military) specifications were a dominant driving force, and most standards were oriented to government policies. The methodology manuals were often policy documents outlining the type of information to be produced by a project but not how to obtain that information."

In order to improve the software development practice in industry, the authors make the following recommendations:

a. More and better computer resources, such as, better computing facility, workstations, screen-oriented syntax-directed editors, programming support tools, should be made available for development.

b. Methods and tools should be evaluated for their utility and usability.

c. Tool support, such as, testing tools and debugging tools, should be built for a common high-level language.

d. PDL processors should support an automated set of metrics that cover the design and coding process.

e. Syntax-directed editors, in which the grammar of the underlying language is built in, would facilitate rapid development of valid programs and add to productivity. Its use should be encouraged.

f. The review process should be improved. The reviews or inspections are a strong part of current methodology and its use can be strengthened by use of automated tools.

g. Incremental development such as iterative enhancements should be used.

GLOSSARY

AFR	Air Force Regulation
AIE	Ada Integrated Environment
AJPO	Ada Joint Program Office
ALS	Ada Language System
APSE	Ada Programming Support Environment
AR	Army Regulation
C^3	Command, Control, and Communications
C^3I	Command, Control, Communications, and Intelligence
CAIS	Common APSE Interface Specification
CRMP	Computer Resources Management Plan
DARCOM	Department of the Army Development and Readiness Command
DoD	Department of Defense
DoDD	DoD Directive
DIANA	Description Intermediate Attributed Notation for Ada
GE	General Electric
IEEE	Institute of Electrical and Electronics Engineers
LCSS	Life Cycle Software Support
MDG	Module Design Guide
MENS	Mission Elements Needs Analysis
MIL	Military
MIL-STD	Military Standard
PDL	Program Design Language
PSL/PSA	Problem Statement Language/Problem Statement Analyzer
SECNAVINST	Secretary of the Navy Instruction
SREM	Software Requirements Engineering Methodology
U. S.	United States
WWMCCS	Worldwide Military Command and Control System
WIS	WWMCCS Information System

REFERENCES

1. "DoD Digital Data Processing Study," Proc. EIA Fall Symposium--DoD Electronics Forecast, Los Angeles, CA, 1980.

2. P. Wegner, "Capital Intensive Software Technology," IEEE SOFTWARE, Vol. 1, No. 3, pp. 7-46, July 1984.

3. U. S. Department of Defense, Major System Acquisitions, DoD Directive 5000.1, Washington, DC, 18 January 1977.

4. U. S. Department of Defense, Life Cycle Management of Automated Information Systems, DoD Directive 7920.1, Washington, DC, 17 October 1978.

5. U.S. Department of Defense, Management of Computer Resources in Major Defense Systems, DoD Directive 5000.29, Washington, DC, 26 April 1976.

6. U. S. Air Force Logistics Command, Acquisition and Support Procedures for Computer Resources in Systems, AFR 800-14, Vol II, 25 Nov. 1975.

7. U. S. Army Development and Readiness Command (DARCOM), Management of Computer Resources in Army Mission Critical Defense Systems (Preliminary Draft), AR 70-XX, 14 November 1983.

8. Secretary of the Navy Instructions (SECNAVINST), Management of Computer Resources in the Department of the Navy Systems, SECNAVINST 5200.32, 11 Jun 1979.

9. U.S. Department of Defense, Computer Programming Language Policy, DoD Directive 5000.31, Washington, DC, 10 June 1983

10. J. R. Green and W. A. Whitaker, "Ada Foundations for WIS," Proc. Washington Ada Symposium, Association of Computing Machinery, pp. 19-25, March 1984.

11. W. A. Whitaker, "The U. S. Department of Defense Common Higher Order Language Effort," SIGPLAN Notices, p. 2, February 1978.

12. U. S. Department of Defense, Ada Programming Language, ANSI/MIL-STD-1815, February 1983.

13. R. Wiener and R. Sincovec, Software Engineering with Modula-2 and Ada, John Wiley and Sons, New York, NY, 1984.

14. G. Booch, Software Engineering with Ada, The Benjamin/Cummins Publishing Co., 1983.

15. P. Fonash, "Ada Program Overview," SIGNAL, pp. 27-31, July 1983.

16. R. F. Brender and I. R. Nassi, "What is Ada?," IEEE COMPUTER, New York, pp. 17-32, June 1981.

17. Sperry Corporation, Ada-News, Vol. 3, No. 2, June 1984.

18. M. Wolfe, et al., "The Ada Language System," IEEE Computer, pp. 37-45, June 1981.

19. M. J. Ryer, "Experience in Using Ada to Implement a Compiler," Proc. Washington Ada Symposium, pp. 93-97, March 1984.

20. Department of Defense, Ada Joint Program Office, Common APSE Interface Set Specification, CAIS 1.3, 31 July 1984.

21. M. A. Jackson, Principles of Program Design, Academic Press, New York, NY, 1975.

22. K. Orr, Structured Requirements Definition, Ken Orr and Associates, Inc., Topeka, KS, 1981.

23. H. D. Mills, Software Productivity, Little, Brown and Company, Boston, MA, 1983.

24. H. Mills, "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin (ed.), Prentice Hall, Englewood Cliffs, NJ, pp. 42-45, 1971.

25. W. Stevens, G. Meyers, and L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13, No. 2, pp. 115-139, 1974.

26. E. Yourdon, and L. Constantine, Structured Design, Prentice Hall, Englewood Cliffs, NJ, 1979.

27. The MITRE Corporation, Ada-based Design Language, WP-25007, R. F. Hillard II, Bedford, MA, 30 December 1983.

28. The MITRE Corporation, Preliminary Program Manager's Guide to Ada, WP-25012, R. G. Howe, et al., Bedford, MA, 30 November 1983.

29. Institute of Electrical and Electronics Engineers, Inc., Using Ada as a Design Language (draft version 2.2), Software Engineering Standards Committee, New York, August 1984.

30. A. G. Duncan, et al., "Communications System Design using Ada," Proc. Seventh International Software Engineering Conference, IEEE cat. no. 84CH2011-5, pp. 398-407, March 1984.

31. D. L. Parnas, et al., "The Module Structure of Complex Systems," Proc. Seventh Internation Software Engineering Conference, IEEE Cat. No. 84CH2011-5, pp. 408-417, March 1984.

32. S. M. Shatz, "Communications Mechanisms for Programming Distributed Systems," IEEE COMPUTER, Vol. 17, No. 6, pp. 21-29, June 1984.

33. J. E. Fawcette, "Ada Goes to Work," Defense Electronics, Vol. 14, No. 7, pp.60-81, July 1982.

34. S. F. Zigler, "Consideration in Acquiring Ada Compiler," Proc. Washington Ada Symposium, pp. 85-90, March 1984.

35. S. Zigler, and R. Weicker, "Ada Language Statistics for the iMAX-432 Operating System," Ada Letters, pp. 63-67, May-June 1983.

36. G. Persch, M. Dausmann and G. Goos, "Early Experience with Programming Language Ada," Ada Letters, p. 63-67, July 1983.

37. S. N. Mohanty, "Software Cost Estimation: Present and Future," Software Practice and Experience, Vol. 11, pp. 102-121, 1981.

38. S. N. Mohanty, "An Overview of Software Cost Models," IEEE Tutorial: Computer System Design, IEEE Press, New York, NY, Sept. 1984.

39. B. Mathis, presentation at SIGAda Meeting, 30 July - 1 August, Hyannis, MA., 1984.

40. E. M. Prell and A. P. Sheng, "Building Quality and Productivity into a Large Software System," IEEE SOFTWARE, Vol. 1, No. 3, pp. 47-55, July 1984.

41. S. N. Mohanty, "On Verification and Validation," Proc. Symposium on Software Verification and Validation, NTIS-PB1-236465, pp. 1-52, June 1981.

42. S. N. Mohanty, "IV&V: Key to Quality Computer Systems," Proc. Workshop on Product Assurance Techniques for Embedded Computer Systems (eds., D. Smock and S. Mohanty), Naval Surface Weapons Center, White Oak, MD (will be available from NTIS and DTIC), pp. 3-5 - 3-26, January 1984.

43. The MITRE Corporation, Product Assurance Guidelines, MTR-84W00044, J. Hayes and S. Mohanty, McLean, VA, March 1984.

44. M. Zelkowitz, et al., "Software Engineering Practices in the US and Japan," IEEE COMPUTER, Vol. 17, No. 6, pp. 57-66, June 1984.

DISTRIBUTION LIST

INTERNAL

A-10 G. J. MacDonald
C. A. Zraket

D-14 E. C. Brady
A. J. Roberts
G. F. Steeg

W-90 J. W. Benoit
R. E. Bernstein
W. F. Foster
R. M. Harris
L. S. Pocinki
S. Polk
J. K. Summers
J. H. Wood

W-91 F. A. Frangione
R. G. Henderson
S. R. Jones

W-92 N. L. Broome
S. J. Turner

W-93 S. Chokhani (5)
R. B. Dial
R. A. Mikelskas (5)
S. N. Mohanty (25)

Records Resources (2)

EXTERNAL

MAJ A. H. Kopp (15)
Director, Ada Joint Program Office
The Pentagon, Room 3D139 (A/N)
Washington, D.C. 20301

Dr. R. Mathis (15)
Director, STARS Joint Program Office
The Pentagon, Room 3D139 (A/N)
Washington, D.C. 20301

END

FILMED

2-85

DTIC